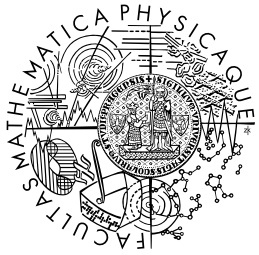


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Peter Grajcar

Generating a drawing according to a textual description

Institute of Formal and Applied Linguistics

Supervisor of the bachelor thesis: Mgr. Rudolf Rosa, Ph.D.

Study programme: Computer Science

Study branch: Programming and Software Systems

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor Mgr. Rudolf Rosa, Ph.D., for his advice and help. Furthermore, I would like to thank my colleagues and friends Katarína Dančejová and Jakub Čatloš for reviewing the text of this thesis and others who helped me throughout the course of my studies.

Title: Generating a drawing according to a textual description

Author: Peter Grajcar

Institute: Institute of Formal and Applied Linguistics

Supervisor: Mgr. Rudolf Rosa, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Text-to-image generators have improved significantly with the recent development of deep neural networks. However, generating complex scenes with multiple objects and relations still remains a difficult problem. In this thesis, we implement a text-to-drawing generator using scene graphs as an intermediate structure. We focus on determining object size and position given a scene graph. We propose a rule-based and classifier-based approach to determine the object position and multiple approaches for size extraction from the scene graph dataset. We provide details of our implementation. We compare and evaluate our approaches and present the results. Finally, we propose potential future use in photorealistic text-to-image generation.

Keywords: natural language processing image generation scene graphs

Contents

Introduction

This thesis aims to implement a drawing generator capable of generating complex scenes for given textual descriptions of the scenes. Generating drawings is substantially less ambitious than generating photorealistic images. However, the problem still poses multiple challenges whose solutions may also be useful in photorealistic image generation.

Despite the significant progress in the field of text-to-image generation, these generators still struggle with more complex scenes. Many approaches only work on limited domains [1]. While promising achievement in a zero-shot text-to-image generation has recently been made [2], generating complex images remains a challenge. One fairly recent approach [3] uses a scene graph representation of the scene in an attempt to address the challenges of complex image generation with many objects and relations. The scene graph is a structure that represents the relations and objects in the form of a directed graph. The scene graph is used as an intermediate structure. However, generating a scene graph from a natural language and generating images from the scene graphs are still complex tasks.

Our work focuses on determining the spatial properties (the position and the size) of the objects within the scene. We propose multiple approaches and compare them. Our approaches may serve as an alternative to box regression methods used in existing scene graph based text-to-image generation [3, 4]. Limiting ourselves to generating drawings allows us to use an existing dataset of hand-drawn images of common objects such as *Quick, Draw!* [5]. We also implement a rule-based description processing that converts text to a scene graph based on the syntactic analysis.

While we are limited to drawings in this thesis, the proposed approaches for determining the positions and sizes could be used for layout generation. The layout is a bounding box corresponding to an object in the scene, which may also be used for image generation [6].

This thesis is structured into six chapters. The first chapter provides an overview of recent text-to-image generators, work related to scene graphs, and drawing datasets. The second chapter introduces our approach for drawing generation. In the third chapter, we provide details of the implementation of our approach presented in the second chapter. The next chapter contains brief instructions for users of our implementation. The fifth chapter presents the results of our approaches for determining spatial properties of objects. The chapter is followed by a discussion of the results and a general conclusion.

1. Related Work

1.1 Text-to-Image Generation

Most of the recent research on text-to-image generators has focused on generating photorealistic images [1, 2, 3, 6, 7, 8, 9, 10] using machine learning approaches such as Generative Adversarial Networks – GANs [1, 6], Cascaded Refinement Networks – CRNs [3] or autoregressive models [2, 8, 9]. However, these models are limited to generating low-resolution images, e.g., 32×32 [7, 8, 9], 64×64 [3, 9], 128×128 [10], 256×256 [1, 2]. Some approaches do not generate images directly from the text but use intermediate structures such as scene graphs [3, 4] or layouts [6].

Figure 1.1: Examples of generated photorealistic images. Top row: Johnson et al. [3], bottom row: Zhao et al. [6].

1.2 Scene Graphs

A scene graph is a graph-based representation of a scene, where vertices of the graph are objects and edges are relations between objects. Scene graphs have been used for image retrieval [11, 12], image generation [3], improving [13] or evaluating [14] image captions. Schuster et al. [11], proposed rule-based and classifier-based approaches for converting sentences to scene graphs with no significant performance difference between the two. We use our own yet similar rule-based approach for text-to-scene graph conversion.

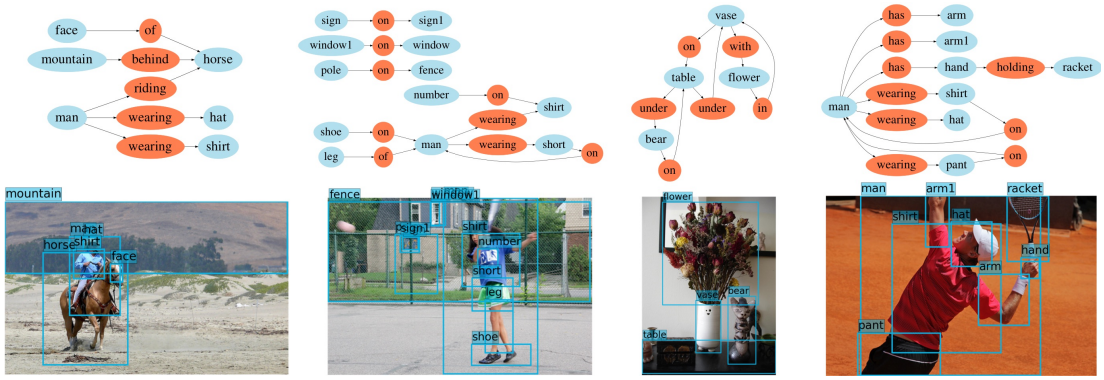


Figure 1.2: Examples of scene graphs from the *Scene Graph* dataset.[1] [15]

Most work on scene graphs is based on *Visual Genome* [16] dataset, which contains scene graphs annotated by humans. This thesis uses *Scene Graph* [15] dataset based on the *Visual Genome*. Unlike the original dataset, this dataset is deprived of ambiguous object names and poor quality bounding boxes.

1.3 Drawing Datasets

Google's *Quick, Draw!* [5] dataset is the largest hand-drawn sketch dataset at the time. It contains 50 million individual drawings classified into 345 categories. The *Quick, Draw!* dataset provides the broadest range of categories compared to other widely used datasets such as *TU-Berlin* [17] dataset with 250 categories and *Sketchy* [18] dataset with 125 categories.

The *Quick, Draw!* [5] dataset contains sketches of common objects represented as sets of pen strokes. Figure 1.3 shows a sample of sketches present in the dataset.

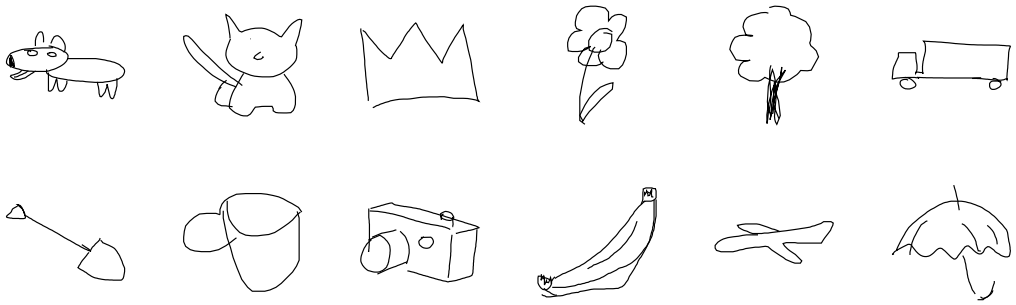


Figure 1.3: Sample of drawings from the *Quick, Draw!* dataset.

[1]Image taken from the *Scene Graph* dataset webpage `https://cs.stanford.edu/~danfei/scene-graph/`.

2. Our Approach

2.1 Overview

This chapter describes the taken approach for generating images from a natural language description. In this approach, we split the task into two by generating a scene graph from the description and subsequently using the scene graph to generate the image. The image generation pipeline is depicted in Figure 2.1 below.

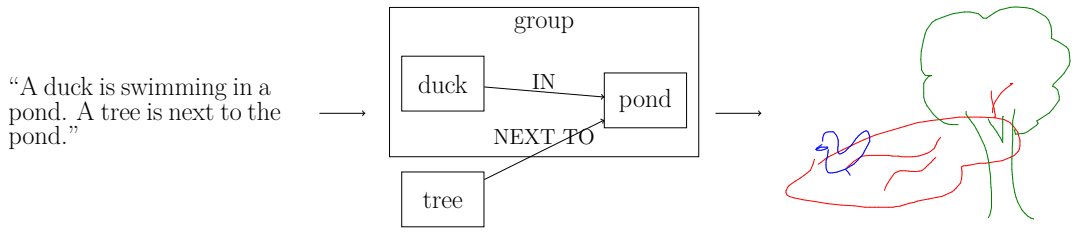


Figure 2.1: Drawing generation pipeline.

2.2 Description Processing

The scene graph is a structure that represents the mutual relations between objects in the scene. Our approach uses syntactic analysis as a shallow semantics parser. Using syntactic analysis, we can extract relations between individual words in the sentences and use these relations to build a scene graph. In the following sections, we discuss the limitations of this approach and present a rule-based algorithm for transforming the syntactic dependency tree into a scene graph.

2.2.1 Limitations

In some cases, the syntactic structure of a sentence does not contain all the necessary semantic information to construct a corresponding scene graph. For instance, consider the two sentences "A man in a car is on the seat" and "A man in a car is on the road". These two sentences have the same syntactic structure (Figure 2.2). In the first sentence, semantically, the preposition "on" describes the relation between the man and the seat. However, in the second sentence, the same preposition puts into relation the whole car (including the man) and the road. We do not attempt to address this problem in our thesis and accept it as a limitation of our approach.

2.2.2 Dependency Tree Parsing

Our approach uses a tree traversal algorithm to extract the objects and relations from the description. This algorithm is applied to a syntactic dependency tree

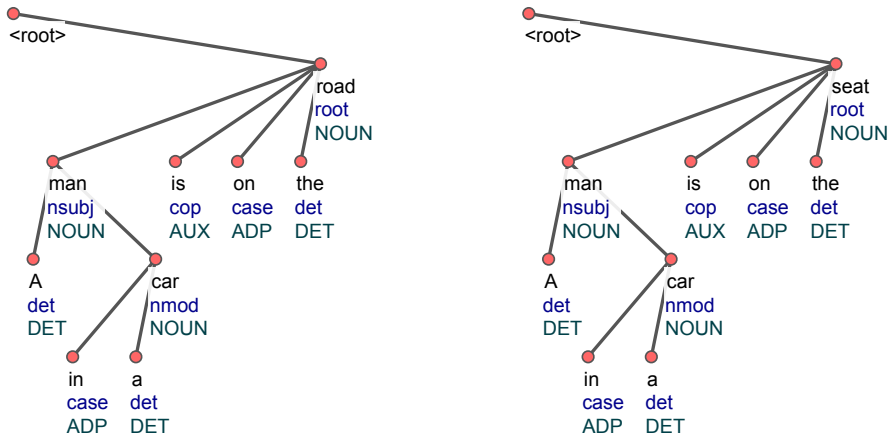


Figure 2.2: Sentences with the same syntactic structure but different semantics.[1]

of the description. In our implementation, we use UDPipe [19] to obtain the dependency trees in *Universal Dependencies version 2* [20] *CoNLL-U* format. The algorithm uses some properties of this format. The implementation is further discussed in Chapter 3 Implementation Details.

The goal of the algorithm is to identify the relations between objects in the dependency tree as semantic subject-predicate-object triples. Note that, unless stated otherwise, whenever we refer to subject-predicate-object triples, we refer to the semantic meaning, not syntactic. The algorithm searches for paths in the dependency tree that contain two nouns — representing the subject and the object — and an adposition representing the predicate. The algorithm can be extended to support verb predicates as well; However, we decided to restrict ourselves to position adpositions. The algorithm traverses the dependency tree in a depth-first manner, processing the nodes from left to right. While it traverses the tree, it maintains a stack of created objects. These objects are created when the algorithm opens a node with a noun. When a node with an adposition is opened, we create a new relation by taking two objects from the object stack.

To allow relations between groups of objects, we replace the object stack with an entity stack. An entity is an abstraction of objects and groups of objects. We keep track of how many objects were created in each subtree. If it is more than one, these objects are removed from the stack to form a group that is added to the top of the stack. Therefore, we can create a relation including a group of objects.

Complex adpositions — adpositions that consist of two or more words — are concatenated into the rightmost dependency tree node. The algorithm distinguishes three complex adposition types of form:

1. adposition-adposition – e.g. inside of
2. adposition-noun-adposition – e.g. in front of
3. adverb/adjective-adposition – e.g. next to

A relation corresponding to the complex adposition is created in the last node; i.e., in the case of *in front of* adposition, dependency tree nodes corresponding to the first two words, *in* and *front*, are skipped.

Similarly, compound nouns such as paper clip, light bulb, pickup truck, and others need to be joined to create only one object. The compound nouns are identified by the *Universal Dependency* [20] relation – `compound`.

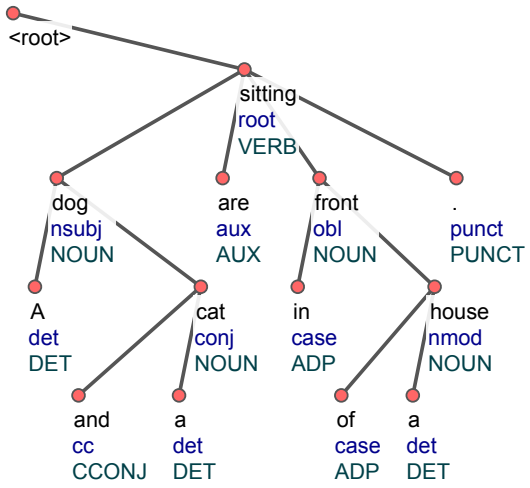


Figure 2.3: Dependency tree of the example descripton.[1]

To illustrate the algorithm, consider the following description: "A dog and a cat are sitting in front of a house." Figure 2.3 shows the syntactic tree corresponding to the description. The nodes of interest are those corresponding to words *dog*, *cat*, *in*, *front*, *of*, and *house*. The nodes are opened and closed by the depth-first algorithm in this order:

1. open *dog*
2. open *cat*
3. close *cat*
4. close *dog*
5. open *front*
6. open *in*
7. close *in*
8. open *house*
9. open *of*
10. close *of*
11. close *house*
12. close *front*

The first two steps create new objects and put them on top of the entity stack. In the third step, the node corresponding to the *cat* is closed, but there are no other entities above the cat on the stack; therefore, the stack remains intact.

[1]Dependency tree illustrations were generated using UDPipe LINDAT service `https://lindat.mff.cuni.cz/services/udpipe/`.

However, when the *dog* node is closed, there is one object on the stack above – the *cat*. The two objects are therefore taken from the stack merged into a group. Next, the *front* node is opened, which is skipped because it is a part of a complex adposition. The same applies for the *in* node. Opening the *house* node adds a new object to the entity stack. The stack now contains two entities – The group and the house on top. In the next step, the node *of* is opened. This word is the last part of the complex adposition *in front of* which means that a new relation is created. A relation between the *house* and the group of two objects – the *dog* and the *cat*.

2.3 Composing a Scene

The next step in our approach is composing a scene from the scene graph. The scene graph can be viewed as a set of constraints on objects and their positions in the scene. We will use the term *scene composition* as a process of determining the size and position of each object in the scene. Methods used for composing a scene that satisfies the constraints are presented later in this section.

2.3.1 Scene Graph Notation

A scene graph is a directed graph with labelled edges which describes a particular scene. Each vertex corresponds to an object in the scene. Edges and their labels represent relations between the objects. Throughout this chapter, we will use the following notation for the relation "object a is in relation p with object b":

$$a \xrightarrow{p} b \tag{2.1}$$

For instance, $dog \xrightarrow{under} tree$ or $book \xrightarrow{on} table$. We will also refer to a, b, and p as subject, object, and predicate, respectively.

2.3.2 Determining the Object's Position

To compose a scene, we need to determine where to put each object. Object positions have to satisfy the constraints given by the scene graph. In this section, we attempt to formally define the constraints and propose methods for creating and satisfying these constraints.

2.3.3 Positional Constraints

We can think of the object positions as x and y coordinates on a canvas. Thus, we can define the positional constraints as a function of points in the two-dimensional plane with two possible outcomes: 1 if the constraint is satisfied and 0 if it is not satisfied. We will denote a constraint function corresponding to a relation $a \xrightarrow{p} b$ as $C^p_{a,b}$. The constraint function is then defined as:

$$C^p_{a,b} \colon \mathbb{R}^2 \to \{0, 1\} \tag{2.2}$$

Suppose we have a constraint function for each relation in the scene graph. Placing an object a within a scene means finding point (x_a, y_a) such that Equation (2.3) holds for every object b_i which is in relation $a \xrightarrow{p_i} b_i$.

$$C^{p_i}_{a,b_i}(x_a, y_a) = 1 \tag{2.3}$$

However, such a point may not exist. Its existence depends on the constraint functions. Therefore, we may accept positions that satisfy a certain portion of the constraints, not necessarily all of them. The approach described in this thesis uses a Monte Carlo algorithm to find a point that satisfies the most constraints on an object.

The Monte Carlo algorithm generates N random normally distributed points $(x_0, y_0), \ldots, (x_N, y_N); x_i \sim \mathcal{N}(x_b, w_b), y_i \sim \mathcal{N}(y_b, h_b)$ where (x_b, y_b) and (w_b, h_b) denote the centre of the object and size of object b respectively. Each of the points is tested using all constraint functions associated with objects a and b. The first point that satisfies the largest number of constraints is designated as the position of object a. This algorithm requires prior knowledge of the position of object b, which requires the existence of topological ordering of the scene graph.

2.3.4 Rule-Based Constraints

In the first approach, we define 5 elementary constraint functions and combine these functions to define constraints for selected adpositions. The five constraint functions are:

1. *On* constraint – Tests whether a point is near the top of the object to which the constraint relates.
2. *Side* constraint – Defines a half-plane; all points in the half-plane satisfy the constraint.
3. *Box* constraint – Tests whether a point lies within a box.
4. *Inside* constraint – Treats the object it relates to as a polygon. A point satisfies this constraint if it lies inside the polygon.
5. *Disjunction* constraint – A composite constraint that encapsulates two or more other constraints. The constraint is satisfied if at least one of the encapsulated constraints is satisfied.

Using these elementary constraints we define constraint functions for a small set of adpositions – *in*, *inside*, *inside of*, *on*, *under*, *below*, *above*, *behind*, *in front of*, and *next to*. Table 2.1 shows which elementary constraints are used to define the constraints for the listed adpositions. Section 3.3.1 describes how each of the elementary constraints is implemented and provides details on how are these constraints combined.

2.3.5 Classifier-Based Constraints

The other approach uses a binary classifier trained on the *Scene Graph* [15] dataset. The classifier matches the constraint function definition 2.2. Given coordinates (x, y), subject a, object b, and predicate p, the classifier decides whether the coordinates (x, y) satisfy the constraint given by $a \xrightarrow{p} b$.

Adposition	**Elementary constraint**
in	*Inside* constraint
inside	*Inside* constraint
inside of	*Inside* constraint
on	*On* constraint
under	*Side* constraint
below	*Side* constraint
above	*Side* constraint
behind	*Box* constraint
in front of	*Box* constraint
next to	*Disjuction* constraint (containing two *side* constraints)

Table 2.1: Rule-based constraints.

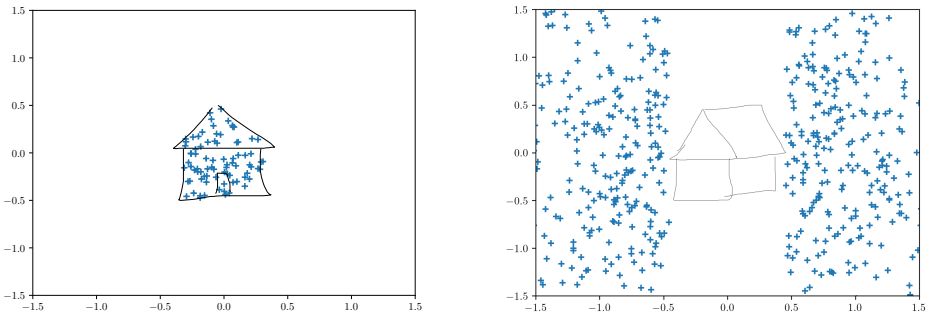


Figure 2.4: Monte Carlo algorithm finds points that satisfy given constraints. *Inside* constraint on the left, *side* constraint on the right.

The classifier has four inputs – subject, predicate, x coordinate, and y coordinate. The subject is encoded as a 100-dimensional vector using word embedding. The *Scene Graph* dataset [15] contains a limited number of predicates. We use the 50 most frequent ones; therefore, the predicate is encoded using a 50-dimensional one-hot vector. Each of the x and y coordinates is encoded as a single float number. Note that we use normalised coordinates relative to the centre of the subject – the absolute x and y coordinates are converted to relative ones and divided by subjects' width and height, respectively. In total, the classifier takes a 152-dimensional input. In Section 5.2, we also show results obtained with a classifier that only takes the predicates and the coordinates as inputs.

The classifier itself is a multilayer perceptron with three hidden layers and a total of 600 hidden units with ReLU activations. We use Stochastic Gradient Descent (SGD) as an optimiser and a cross-entropy loss.

The source of the training data is the *Scene Graph* dataset [15]. The dataset contains the semantic subject-predicate-object triples as well as sizes and positions of the subjects and objects. The positions are absolute. We convert them to the relative position format described in the preceding sections. However, this dataset only contains correctly placed objects. To be able to train the classifier, we need to synthesise data that contains incorrectly placed objects. We do so by replacing the relative position of a subject-object-predicate triple with a position of another triple with a different predicate.

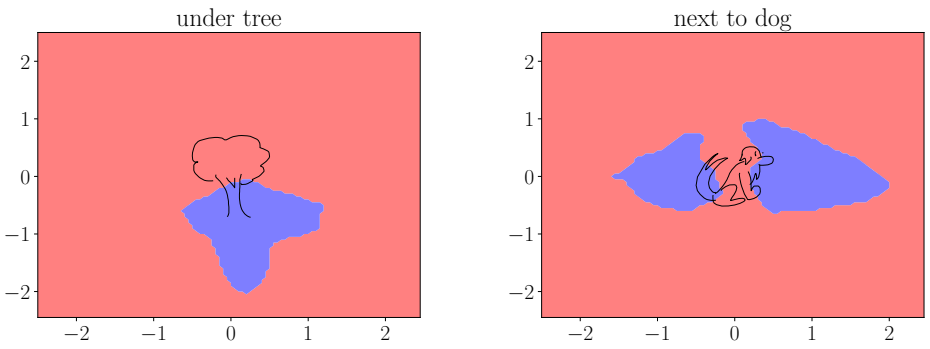


Figure 2.5: Visualisation of the classifier's decision boundaries.

2.3.6 Determining the Object's Size

Besides the object positions within a scene, we also need to determine their sizes. The *Quick, Draw!* [5] dataset contains 345 categories. The relatively small number of categories makes it possible to define the absolute size for each category by hand. This is, in fact, our first approach for determining the object sizes. The objects in the scene can be scaled accordingly to the ratio between the defined sizes. In addition to the process being tedious, this approach also has one apparent flaw. The size ratio of objects often depends on context. For instance, the size ratio of a television and a car is different in the following sentences: "A car is on the television" and "The television is near the car.".

The mentioned problem of absolute sizes suggests a different approach. Instead of defining the absolute size for each category, we can define a relative size between a pair of categories. However, defining relative sizes for 119 025 pairs of categories by hand is infeasible. Instead, we once again use the *Scene Graph* [15] dataset to extract the relative sizes. The following list summarizes three methods used to determine the objects' sizes:

1. Absolute – For each *Quick, Draw!* category, we extract absolute sizes of matching objects' bounding boxes from the dataset and use their average as a size.
2. Relative – If both subject and object are valid *Quick, Draw!* categories, we compute the ratio of their widths and heights. We average these ratios across the whole dataset.
3. Relative + word embedding – Modification of the previous method where we do not require an exact match with a *Quick, Draw!* category. Instead, we add the ratios to the most similar categories. The most similar categories are determined by the cosine similarity of their word embeddings. We also set different similarity thresholds.

Not all the *Quick, Draw!* category pairs are present in the *Scene Graph* dataset. To address this problem, the data obtained using the methods mentioned above can be completed by transitive closure. We estimate the unknown ratio $a : b$ between objects a and b as

$$\frac{a}{b} = \frac{1}{n} \sum_{i=0}^{n} \frac{a}{k_i} \cdot \frac{k_i}{b}$$

where $a : k_i$ and $k_i : b$ are known ratios.

2.4 Image Generation

With a composed scene, the image generation itself is a straightforward process. Both position and size are known for all objects in the scene. The only thing left to do is to select a drawing of the object and draw it onto a drawing canvas. The drawings are randomly selected from the *Quick, Draw!* [5] dataset. We do not use the entire dataset as it also contains irrelevant and inappropriate data. Only a manually selected portion of the dataset is used.

3. Implementation Details

This chapter provides details about the implementation of the approach proposed in Chapter 2. The implementation is written in Python 3. Its source code can be found in Attachment A.1.

3.1 Overview

The implementation consists of three main components: a description processor, a scene composer, and a renderer. These components are depicted in Figure 3.1. The components and transitions between the components correspond to the drawing generation pipeline presented in Chapter 2.

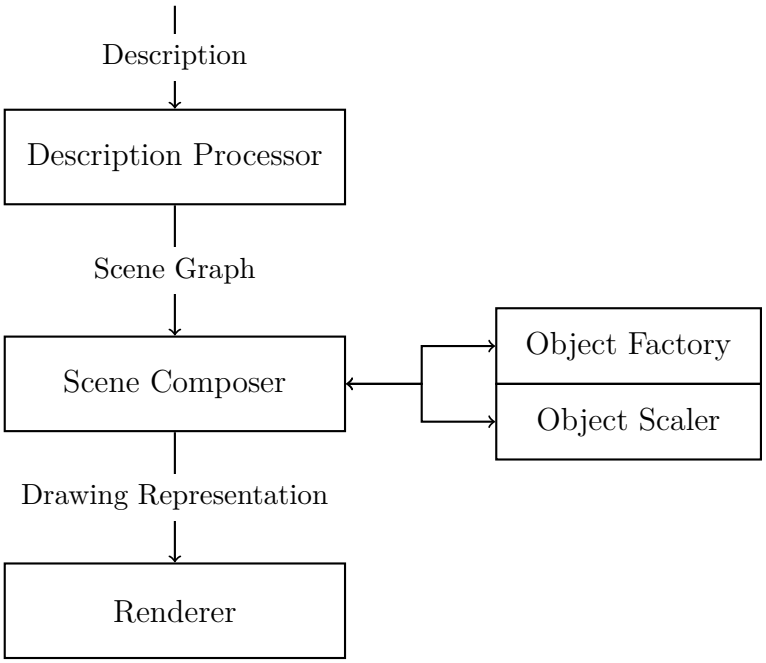


Figure 3.1: Implementation diagram.

Besides the three main components, the diagram in Figure 3.1 also contains two other components: an object factory and an object scaler. Together with the scene composer component, these two components implement the algorithms proposed in Section 2.3 Composing a Scene. Each of the components is discussed in more detail in the following sections.

3.2 Description Processor

The description processor is a component responsible for converting the input text into a scene graph. Description processor has one public method `process`. The method has the following Python signature:

```
process(self, text: str) -> Scene
```

Attachment A.1 contains one implementation of the description processor - `UDPipeDescriptionProcessor`. This implementation uses UDPipe [19] for syntactic analysis. The dependency tree traversal algorithm was described in the Section 2.2 Description Processing. The `resources/predicates` file contains a list of allowed predicates which can be processed by the processor.

The `Scene` class represents a scene graph. It contains two types of entities: `Object` and `Group` entities (subclasses of `Entity`). Each entity has a list of relations represented by `Relation`.

The `Group` entity is just a convenience class which encapsulates other entities. Every group's relation is redirected to each entity within the group. We can extend the notation introduced in Section 2.3.1 Scene Graph Notation, such that $[a, b]$ will denote a group of objects a and b. Then $[a, b] \xrightarrow{p} c$ will effectively become $a \xrightarrow{p} c$ and $b \xrightarrow{p} c$. Similarly $a \xrightarrow{p} [b, c]$ will become $a \xrightarrow{p} c$ and $a \xrightarrow{p} c$.

Apart from the relations, the `Object` class also contains the attribute `word`. This attribute holds a noun that describes the object. Note that in the case of composite nouns, this attribute may contain multiple words.

3.3 Scene Composer

The scene composer determines the position of every object in the scene. The scene composer is a class with the following method:

```
compose(self, scene: Scene) -> List[PhysicalObject]
```

`ConstraintComposer` is an implementation of the algorithm described in Section 2.3.3 Positional Constraints. The `ConstraintComposer` associates each object in the scene with a corresponding `Constraint` object. Then it uses these constraints to determine the positions using the described Monte Carlo algorithm.

The `Constraint` is an abstract callable class. As opposed to the constraint function defined in Section 2.3.3 Positional Constraints, the `Constraint` object operates on a batch of coordinates. It returns a Boolean array that masks the coordinates which satisfy the constraint. Using the notation from Section 2.3.1, we can describe the Python `Constraint` call method:

```
__call__(self, xs: np.ndarray, ys: np.ndarray) -> np.ndarray
```

As a mapping between two vectors:

$$[(x_0, y_0), \ldots, (x_n, y_n)] \mapsto [C^p_{a,b}(x_0, y_0), \ldots, C^p_{a,b}(x_n, y_n)] \tag{3.1}$$

The scene composer generates a single set of random coordinates for one object. These coordinates are applied to every constraint associated with the object. The resulting boolean arrays are summed as integer arrays. By applying $argmax$ to this array, we find an index of coordinates that satisfy most of the constraints. These coordinates are used as the object's position. In other words, for a fixed object a, we set its position to be (x_i, y_i), where:

$$i = \operatorname{argmax} \sum_{(b,p)} [C^p_{a,b}(x_0, y_0), \ldots, C^p_{a,b}(x_n, y_n)] \tag{3.2}$$

3.3.1 Constraints

Section 2.3.4 lists a set of elementary constraints. In this section, we take a closer look at these constraints, describe how they are implemented and how they are combined to define constraints for specific adpositions. All constraints are implemented as subclasses of `Constraint` which was described in the previous section. The constraints are initialised with an object and predicate that relates to the object. Information about the object and predicate are used to determine whether a point in plane satisfies the constraint.

The `OnConstraint` takes the upper 25% of the object's drawing strokes. The constraint tests if a point is within a certain distance from these strokes. The distance is specified by attribute `limit`.

The `SideConstraint` defines a half-plane. All points in this half-plane satisfy the constraint. The constraint has two attributes – `direction` and `offset`. The `direction` attribute is a vector normal to the half-plane boundary. The `offset` attribute moves the boundary further away from the object.

The `BoxConstraint` is satisfied whenever a point lies inside a bounding box of the object to which the constraint relates. In addition, the implementation also contains an attribute `scale` that can scale the bounding box up or down.

The `InsideConstraint` tests whether a point lies within an area spanned by the object it relates to. The area is given by strokes of drawing associated with the object. We consider each stroke to be a polygon. We test whether a point lies inside a polygon defined by one of the strokes. Some strokes contain many line segments, which could lead to a slow evaluation of the constraint. Hence, we apply the Ramer-Douglas-Peucker [21] algorithm to the drawing to reduce its number of line segments.

The primary purpose of `DisjunctionConstraint` is to combine other non-overlapping constraints. The constraint is satisfied when any of the constraints it encapsulates is satisfied. Since we assume the encapsulated constraints are non-overlapping, only one of the constraints can be satisfied at a time. The same could be achieved by specifying more constraints for a single predicate. However, using the disjunction constraint does not require evaluating all the constraints – the evaluation ends after the first satisfied constraint. The non-overlapping regions can be seen in the right image of Figure 2.4.

Table 3.1 below shows how the elementary constraints are used in the rule-based approach to form constraint functions for adpositions listed in Section 2.3.4.

`ClassifierConstraint` implements the classfier-based approach described in Section 2.3.5 Classifier-Based Constraints. The classifier uses *scikit-learn's* [22] multi-layered perceptron classifier trained on the *Scene Graph* dataset. The trained model is loaded from `resources/sklearn/constraints.model` file. All relevant details about the model are discussed in Section 2.3.5.

3.3.2 Object Factory

The object factory is a class with a method with the following Python signature:

```
get_physical_object(self, obj: Object) -> PhysicalObject
```

Adposition	Elementary constraint
in	`InsideConstraint()`
inside	`InsideConstraint()`
inside of	`InsideConstraint()`
on	`OnConstraint()`
under	`SideConstraint(direction=(0, 1))`
below	`SideConstraint(direction=(0, 1))`
above	`SideConstraint(direction=(0, -1))`
behind	`BoxConstraint()`
in front of	`BoxConstraint()`

next to	`DisjunctionConstraint([`
	`  SideConstraint(direction=(-1, 0)),`
	`  SideConstraint(direction=(1, 0))`
	`])`

unknown	`DisjunctionConstraint([`
	`  SideConstraint(direction=(-1, 0)),`
	`  SideConstraint(direction=(1, 0)),`
	`  SideConstraint(direction=(0, 1)),`
	`  SideConstraint(direction=(0, -1)),`
	`])`

Some constructor arguments are omitted for brevity.

Table 3.1: Rule-based constraints implementation.

`QuickDrawObjectFactory` instantiates a `PhysicalObject` using Quick, Draw! [5] Data. It uses a `WordEmbedding` class for resolving objects which are not present in the *Quick, Draw!* categories.

`WordEmbedding` class is a wrapper around a *fasttext* [23] model. It has a single method `most_similar_word(self, word: 'str') -> 'str'` which finds the most similar word from a list of words based on the cosine similarity. The list of words is, in this case, a list of *Quick, Draw!* categories.

3.3.3 Object Scaler

The object scaler determines the object sizes. The object scaler class has the following Python signature:

```
scale(self, sub: PhysicalObject, obj: PhysicalObject,
            pred: str) -> float
```

This method scales the subject `sub`. The size estimation may use the context given by the object `obj` or the predicate `pred`. The size estimation strategy depends on the implementation. We provide two object scaler implementations – `AbsoluteObjectScaler` and `RelativeObjectScaler`. These clases implement the object scaling methods described in Section 2.3.6 Determining the Object's Size.

The `AbsoluteObjectScaler` scales the `sub` object according to a table of absolute (hand crafted) sizes located in `resources/quickdraw/attributes.csv`. The file contains a width and/or height for the given *Quick, Draw!* category.

The `RelativeObjectScaler` takes into account both subject and object (`sub` and `obj`). The subject is scaled according to a table of relative sizes located in `resources/quickdraw/attributes_relative.csv` file. The file contains width and height ratios for the given pair of *Quick, Draw!* categories. The size of `obj` has to be known before calling the `scale` method. Its size is multiplied by the ratio specified in the file to determine the subject's size.

3.4 Renderer

The renderer is the last component in the system. The only purpose of the renderer is to render a final drawing from a `PhysicalObject` list. Two different renderers are provided. The first one, `SimpleRenderer`, renders a PNG image. This renderer is used in the command-line interface. The other renderer has a form of a web application written in JavaScript. The composed scene served over HTTP is rendered onto an HTML canvas. The web interface is discussed in more detail in Section 4.4 Web Interface.

4. User Manual

4.1 Prerequisites

Make sure that the following are installed on your system before proceeding to the installation:

- GNU Make
- Python 3

4.2 Installation

All Python dependencies and required pre-trained models can be downloaded and installed by running the following command in `drawtomat/` directory:

```
make install
```

4.3 Command-Line Interface

The command-line interface can be started by

```
make run
```

or alternatively:

```
export PYTHONPATH=src
python3 -m drawtomat [-h] [--help]
                     [--description DRAWING_DESCRIPTION]
                     [--graph_output GRAPH_OUTPUT_PATH]
                     [--image_output IMAGE_OUTPUT_PATH]
                     [--sizes {absolute,relative}]
                     [--constraints {rule,classifier}]
                     [--show]
```

If the `--description` flag is not set, the program takes the description from the standard input. A path to a generated drawing can be set by `image_output`. The default is `./drawing.png`. If the `--show` flag is set, the image is opened immediately after it is generated. If the `--graph_output` flag is set, a scene graph will be saved to the specified file in the Graphviz DOT language. Note that when running the program without Make, one needs to activate the Python virtual environment beforehand. Options `sizes` and `constraints` allows user to choose the approach for determining object size and position.

4.4 Web Interface

4.4.1 Server

The implementation contains a Flask web server, which provides a simple HTTP API. The server can be started by running the following command:

```
make run-api
```

The API has a single endpoint `POST /drawtomat`. The format of request and response bodies are described by tables 4.1 and 4.2. Both tables specify JSON formats accepted and provided by the API.

Key	Type	Description
`description`	`string`	A description of a drawing (limited to 1000 characters).
`options`	`object`	Drawing generation options.
`options.constraints`	`string`	Allowed values are `absolute` or `relative`. If the value is equal to `"classifier"`, positions of objects in the scene will be determined using the classifier-based approach.
`options.sizes`	`string`	Allowed values are `"rule"` or `"classifier"`. If the value is equal to `"relative"`, sizes of objects in the scene will be determined based on size ratios extracted from *Scene Graph* dataset.

Table 4.1: Request JSON format.

Key	Type	Description
`description`	`string`	A description of a drawing.
`bounds`	`object`	Defines a bounding box of the drawing.
`bounds.top`	`number`	The maximum of all y coordinates.
`bounds.bottom`	`number`	The mininum of all y coordinates.
`bounds.right`	`number`	The maximum of all x coordinates.
`bounds.left`	`number`	The minimum of all x coordinates.
`drawing`	`array`	Drawing representation similar to *Quick, Draw!* format. The drawing is represented as an array of objects in the scene. Objects are arrays of strokes. Each stroke is an array itself containing three sequences - x coordinates, y coordinates and time in milliseconds.

Table 4.2: Response JSON format.

Sample Request

```
POST /drawtomat HTTP/1.1
Host: localhost:5000
```

```
Content-Type: application/json

{
    "description": "A dog is sitting under a tree.",
    "options": {
        "constraints": "classifier",
        "sizes": "relative"
    }
}
```

Sample Response

```
HTTP/1.1 200 OK
Content-Type: application/json
Server: Werkzeug/1.0.1 Python/3.8.6
Date: Thu, 11 Mar 2021 09:37:29 GMT

{
    "bounds": {
        "bottom": 197.13186109372901,
        "left": -187.71175260273606,
        "right": -10.021259329191409,
        "top": 442.4159101648598
    },
    "description": "A dog is sitting under a tree.",
    "drawing": [
        [
            [
                [-84.61983725860478, ..., -69.03160196448714],
                [268.89656697608194, ..., 345.9553905054937],
                [0, ...,  553]
            ],
            ...
        ]
    ]
}
```

4.4.2 Client

The web client can be found in `web/` directory, which contains a single HTML page `index.html`. The webpage contains a text field for the drawing description and a canvas where the drawing is drawn. A collapsible menu is under the text field and can be toggled by clicking on "Advanced". In the menu, the user can set the drawing speed and strategies for position and size determination. The server address can be configured in `config.js` file. The web client is depicted in Figure 4.1.

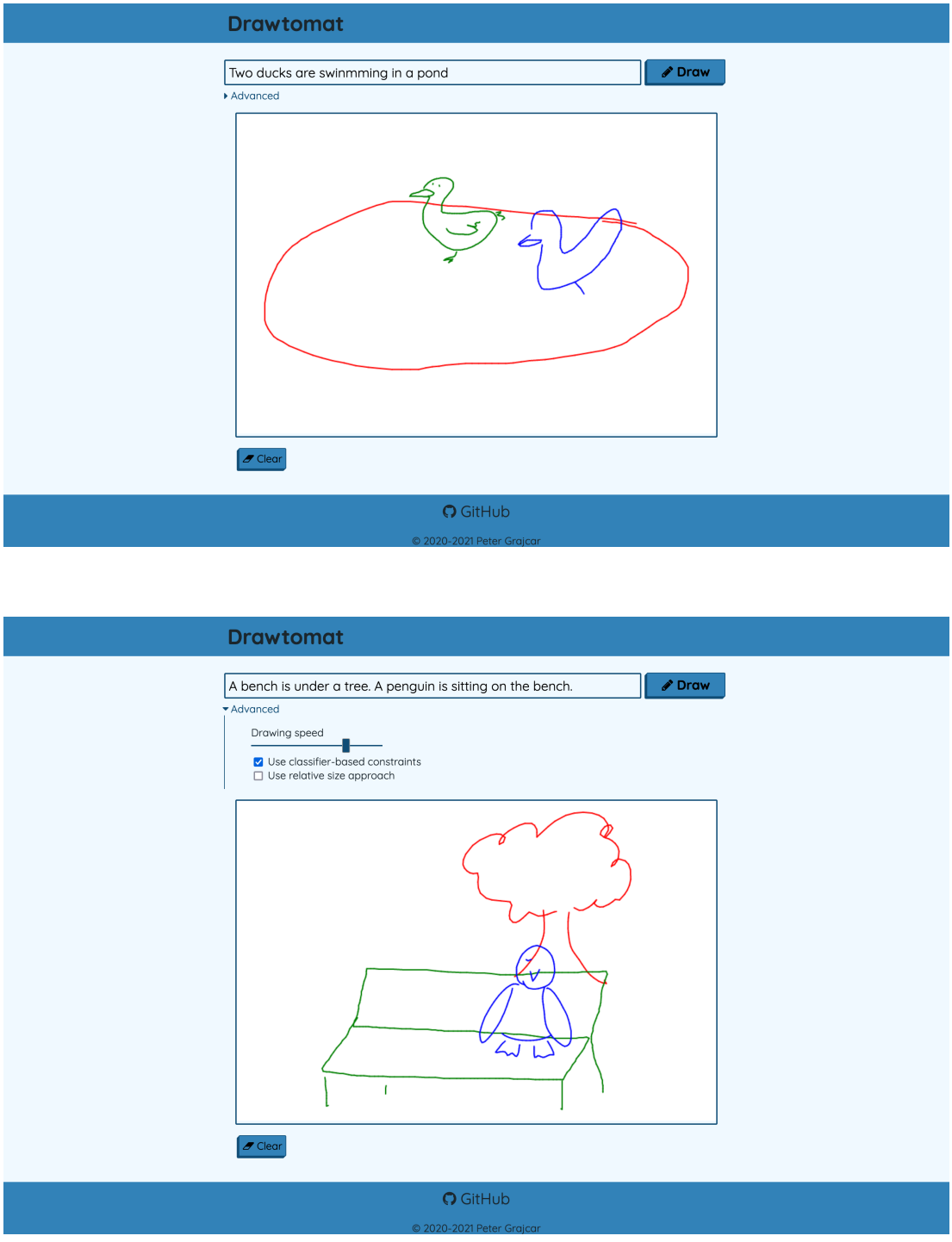


Figure 4.1: The web client.

5. Results

5.1 Extracting Object Sizes

In Section 2.3.6 Determining the Object's Size, we proposed three methods for determining object size by extracting data from the *Scene Graph* [15] dataset. In this section, we compare and evaluate the three methods.

5.1.1 Evaluation

The size data extracted from the *Scene Graph* [15] dataset using the above method are compared against the hand-written sizes. The extracted absolute sizes can be directly compared to the hand-written data. The relative sizes are compared against size ratios between the absolute hand-written sizes.

We use the Root-Mean-Square Error (RMSE) as a metric. For N predictions $y_0, \ldots, y_n$ and target values $t_0, \ldots, t_n$, RMSE is defined as:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(y_i - t_i)^2}$$

The second metric we are interested in is the percentage of covered pairs, i.e., for how many pairs, out of the total of 119 025 pairs, we were able to extract the size data.

5.1.2 Quantitative and Qualitative Results

The RMSEs of the proposed size extraction methods are reported in Tables 5.1 and 5.2. Table 5.1 compares variants of the proposed methods with transitive closures.

Method	Width RMSE	Height RMSE	Pairs Covered
Absolute	51.93	43.17	**91.49%**
Relative	**12.97**	16.46	7.50%
Relative + word embedding	43.93	42.46	63.60%
Relative + word embedding (0.85)	19.78	24.87	10.34%
Relative + word embedding (0.95)	13.13	**16.12**	7.92%

Table 5.1: Comparison of the size extraction methods.

All relative size methods outperformed the absolute size method in terms of RMSE. However, these methods cover significantly fewer pairs. The transitive closure partially solves the small pair coverage problem, but it also introduces larger errors. Similarly, the word embeddings helped to increase both the pair coverage and errors.

Figure 5.1 compares the absolute size and relative size methods on the example given in Section 2.3.6. In this particular example, the relative size method addresses the problem stated in the mentioned section. However, the rightmost

Method	Width RMSE	Height RMSE	Pairs Covered
Absolute*	51.93	43.17	91.49%
Relative	**43.65**	**41.04**	79.70%
Relative + word embedding	55.53	47.92	**100%**
Relative + word embedding (0.85)	53.35	45.65	86.57%
Relative + word embedding (0.95)	49.39	42.03	82.30%

* Transitive closure cannot be applied to the absolute sizes. The results are the same as in Table 5.2. For the sake of comparison, they are also included in this table.

Table 5.2: Comparison of the size extraction methods with transitive closure.

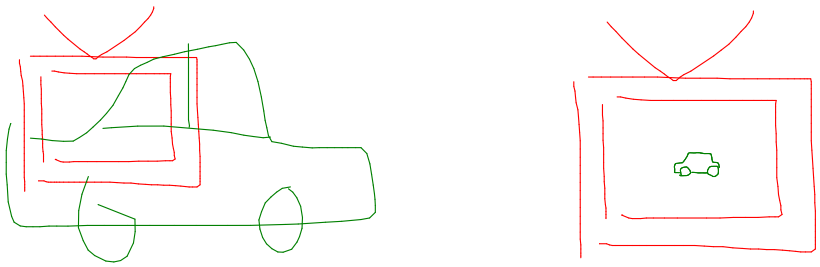


Figure 5.1: Drawings generated from description "A car is on the television." using absolute (right) and relative (left) sizes.

drawing in Figure 5.2 shows that the method fails the other way around. Presumably, the dataset captures certain pairs of objects only in a narrow variety of contexts, leading to biased estimations. Furthermore, the relative size approach is oblivious to the predicates associated with the relation. We have chosen not to include the predicate as it would vastly decrease the pair coverage.

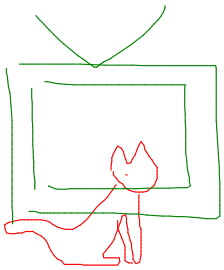
5.2 Positional Constraints

In Section 2.3.4 and Section 2.3.5, we introduced a rule-based and a classifier-based approach for determining the objects' positions. In this section, we compare those two approaches.

The first version of the classifier-based approach considers only the predicate and the relative position described in Section 2.3.5. As shown in Figure 5.3, constraints implemented using this classifier closely resemble the rule-based alternatives. It shows that the *Scene Graph* [15] dataset is a viable source of semantic information about the connection of predicates and mutual positions of objects. The figure also suggests that this classifier is a sufficient replacement for the simple rule-based approach.

The second version of the classifier takes into account the semantic subject. Figure 5.4 shows that this classifier can capture different semantics of a predicate with respect to the subject. More examples are shown in Figure 5.5.

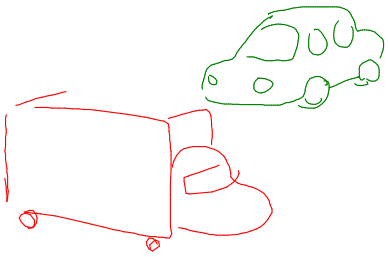
The major drawback of the classifier-based approach is the lack of information about the shape of the object. This problem is partially illustrated by Figure 5.6.



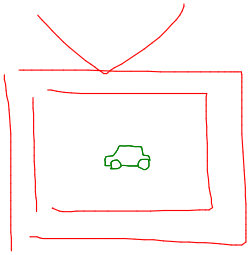
“A television is behind a cat.”

“A cat is sitting on a television.”

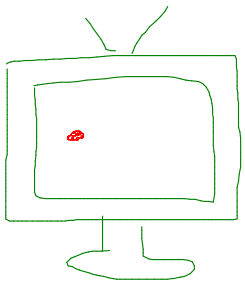
“A truck is behind a car.”



“A car is next to a truck.”



“A car is on the television.”



“A television is in front of a car.”

Figure 5.2: Drawings generated using the relative size method.

In some cases, the rule-based approach provides more precise boundaries than the classifier-based alternative.

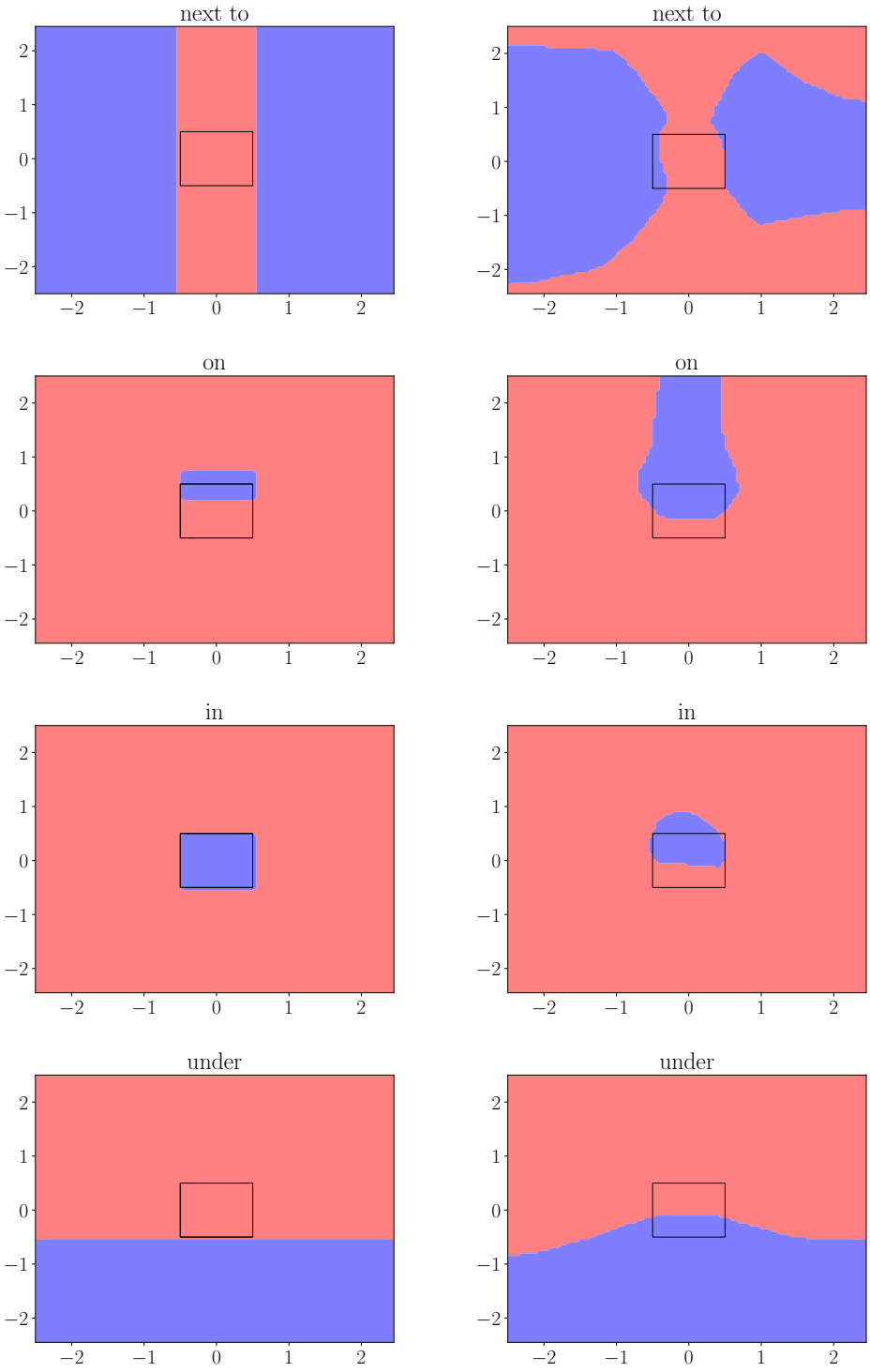


Figure 5.3: Rule-based constraints (on the left) compared to the classifier-based constraints (on the right). The classifier used in these examples was trained using only predicates and the x, y coordinates.

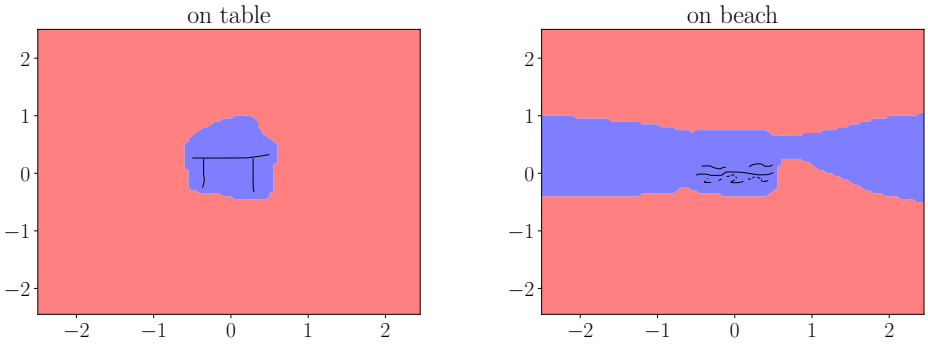


Figure 5.4: Different decision boundaries capture semantic difference of the same predicate in different contexts.

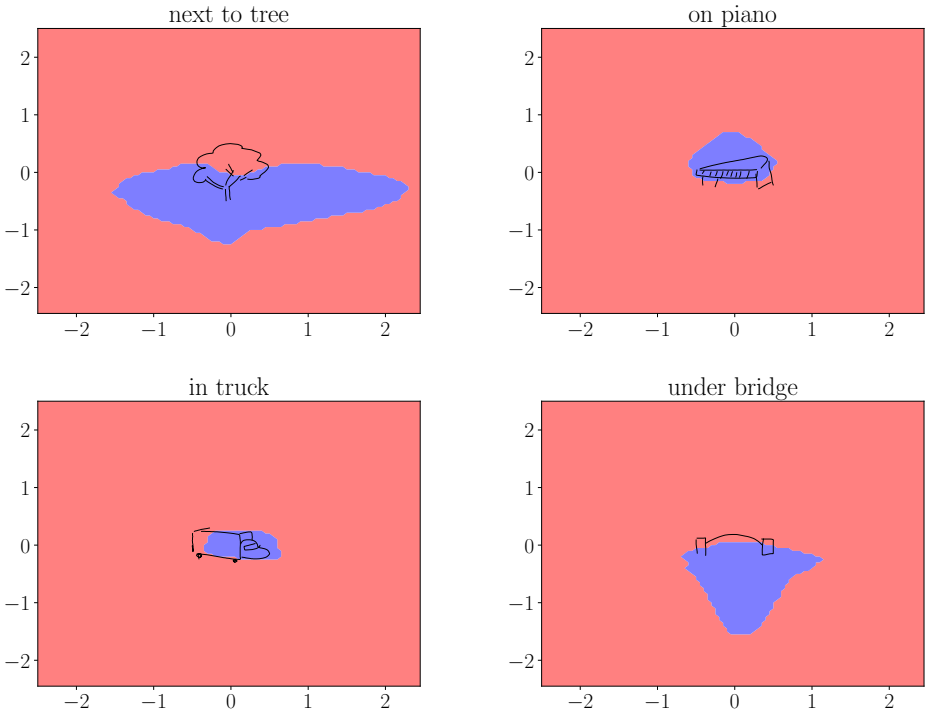


Figure 5.5: More examples of the classifier-based constraints.

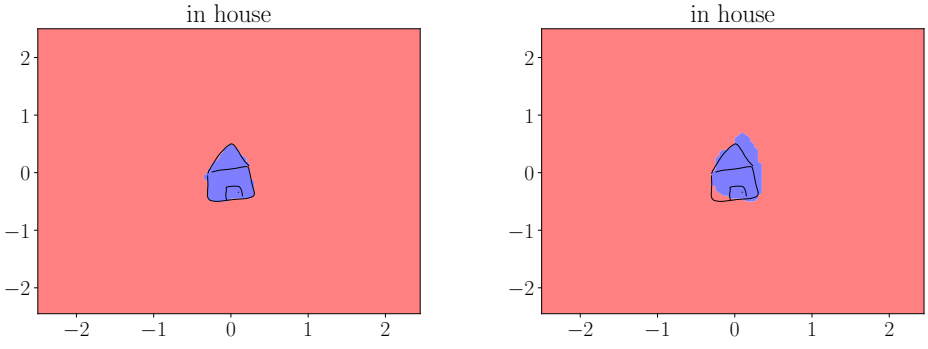


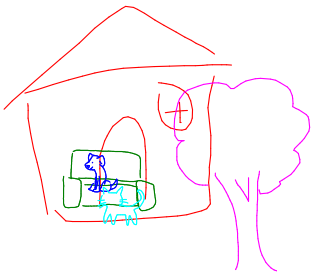
Figure 5.6: The rule-based *inside* constraint (on the left) can be more precise than the classifier-based one (on the right).

6. Discussion

None of the approaches compared in the preceding chapter is flawless, and each works better under certain circumstances. Our implementation gives the user an option to choose the approach and experiment with the results. Figure 6.1 shows that in many cases, the results are similar across different selected approaches. The random nature of the Monte Carlo algorithm allows generating multiple different drawings for one description. This randomness increases the chances that the generated image will eventually correspond with the input description. Future research might focus on automatic scoring and cherry-picking of the generated scenes. Other possible improvements may be achieved by employing more advanced machine-learning models used for constraints and size prediction. Our implementation relies on a simple rule-based description processing which is sufficient for simple sentences; However, finding a better approach may also be a subject of future research.

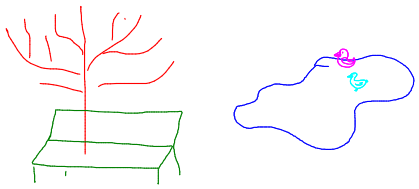
(1) “A dog and a cat are sitting on a couch. The couch is in a house and there is a tree next to the house.”

rule-based constraints, absolute sizes

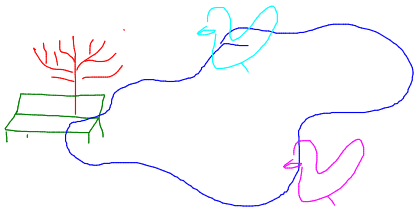


classifier-based constraints, absolute sizes

(2) “Two ducks are swimming in a pond which is next to a tree. a bench is under the tree.”

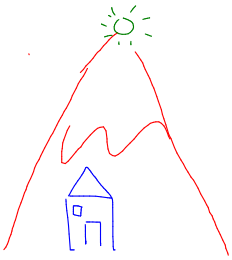


classifier-based constraints, absolute sizes

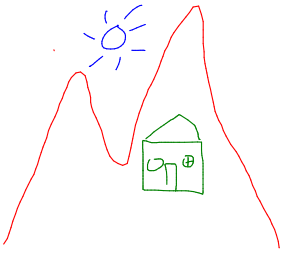


classifier-based constraints, relative sizes

(3) “There is a house, and mountains behind it. The sun is rising above the mountains.”



rule-based constraints, relative sizes



classifier-based constraints, relative sizes

Figure 6.1: Examples of generated drawings. Each example is labelled with the used approach for determining the position and size. The absolute sizes refer to the hand crafted absolute size table.

Conclusion

In this thesis, we have developed a command-line and a web-based application that generates drawings from a textual description using scene graphs as an intermediate structure. We have proposed various approaches for determining the sizes and positions of objects in a scene based on their relations in the scene graph. We have shown that our proposed approach, in many cases, can generate drawings that correspond to the description.

Our approach may find their use in layout generation, used by some photorealistic image generators [6]. Photorealistic image generators that generate images using the scene graphs currently use box regression networks for this task [3, 4]. Our approach could be an alternative to the box regression method. However, comparison with the box regression networks is beyond the scope of this thesis, and it may be a subject of future work.

Bibliography

[1] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris N Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 5907–5915, 2017.

[2] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-Shot Text-to-Image Generation, 2021.

[3] Justin Johnson, Agrim Gupta, and Li Fei-Fei. Image generation from scene graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1219–1228, 2018.

[4] Subarna Tripathi, Anahita Bhiwandiwalla, Alexei Bastidas, and Hanlin Tang. Using scene graph context to improve image generation. *arXiv preprint arXiv:1901.03762*, 2019.

[5] David Ha and Douglas Eck. A neural representation of sketch drawings. *arXiv preprint arXiv:1704.03477*, 2017.

[6] Bo Zhao, Lili Meng, Weidong Yin, and Leonid Sigal. Image generation from layout. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8584–8593, 2019.

[7] Elman Mansimov, Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Generating images from captions with attention. *arXiv preprint arXiv:1511.02793*, 2015.

[8] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. *arXiv preprint arXiv:1606.05328*, 2016.

[9] Aaron Van Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International Conference on Machine Learning*, pages 1747–1756. PMLR, 2016.

[10] Scott Reed, Zeynep Akata, Santosh Mohan, Samuel Tenka, Bernt Schiele, and Honglak Lee. Learning what and where to draw. *arXiv preprint arXiv:1610.02454*, 2016.

[11] Sebastian Schuster, Ranjay Krishna, Angel Chang, Li Fei-Fei, and Christopher D Manning. Generating semantically precise scene graphs from textual descriptions for improved image retrieval. In *Proceedings of the fourth workshop on vision and language*, pages 70–80, 2015.

[12] Justin Johnson, Ranjay Krishna, Michael Stark, Li-Jia Li, David Shamma, Michael Bernstein, and Li Fei-Fei. Image retrieval using scene graphs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3668–3678, 2015.

[13] Siqi Liu, Zhenhai Zhu, Ning Ye, Sergio Guadarrama, and Kevin Murphy. Improved Image Captioning via Policy Gradient optimization of SPIDEr. *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct 2017.

[14] Peter Anderson, Basura Fernando, Mark Johnson, and Stephen Gould. Spice: Semantic propositional image caption evaluation. In *European conference on computer vision*, pages 382–398. Springer, 2016.

[15] Danfei Xu, Yuke Zhu, Christopher Choy, and Li Fei-Fei. Scene Graph Generation by Iterative Message Passing. In *Computer Vision and Pattern Recognition (CVPR)*, 2017.

[16] Ranjay Krishna, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, Michael Bernstein, and Li Fei-Fei. Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations. 2016.

[17] Mathias Eitz, James Hays, and Marc Alexa. How Do Humans Sketch Objects? *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):44:1–44:10, 2012.

[18] Patsorn Sangkloy, Nathan Burnell, Cusuh Ham, and James Hays. The Sketchy Database: Learning to Retrieve Badly Drawn Bunnies. *ACM Transactions on Graphics (proceedings of SIGGRAPH)*, 2016.

[19] Milan Straka. UDPipe 2.0 prototype at CoNLL 2018 UD shared task. In *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 197–207, Brussels, Belgium, October 2018. Association for Computational Linguistics.

[20] Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Jan Hajič, Christopher D Manning, Sampo Pyysalo, Sebastian Schuster, Francis Tyers, and Daniel Zeman. Universal dependencies v2: An evergrowing multilingual treebank collection. *arXiv preprint arXiv:2004.10643*, 2020.

[21] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics*, 5:135–146, 2017.

List of Figures

List of Tables

A. Attachments

A.1 Source Code

Zipped source code that contains:

- implementation of the command-line interface, web server in the `drawtomat` directory;
- implementation of the web client in the `web` directory;
- various experiments and scripts used for training the used models in the `experiments` directory.

The source code is also available as a GitHub repository `https://github.com/peter-grajcar/drawtomat`.